# Privilege Delegation and Agent-Oriented Access Control in Naplet

Cheng-Zhong Xu and Song Fu
Department of Electrical and Computer Engineering
Wayne State University, Detroit, Michigan 48202
{czxu,oaksong}@wayne.edu

## Abstract

*Access control in existing Java-based mobile agents is mostly based on code source due to limitations of early Java security architecture. That is, authorization is based on where the agent code comes from, regardless of the subject of code execution. This paper presents an agent-oriented access control strategy, by taking advantage of the latest Java subject-based security features. It allows the agents to exercise their privileges based on their own roles in distributed applications. The access control is realized by a group of additional agent-oriented permissions. They grant access control privileges to system resources and services in a flexible and secure manner. Permission implications between agents of different clone generations and from different owners are also supported.*

## 1 Introduction

Mobile agent based computing provides an alternative paradigm to the conventional client/server distributed computing model [2]. Until recently, mobile agent systems were developed primarily based on script languages like Tcl and Telescript. Latest proliferation of mobile agent technology is mostly due to the popularity of Java. The platform-neutral Java runtime environment and its dynamic class loading model, coupled with other advanced features like serialization and remote method invocation, greatly simplifies the construction of mobile agent systems. Many systems have been developed, most notably including Aglet [6], Ajanta [9], Concordia [11], D'Agent [4], Odyssey [12], and Voyage [8]. Readers are referred to [10] for an early review. Recently, we developed a lightweight mobility system, Naplet, in support of adaptive Internet services [7][13][5][14].

A critical issue in the construction of mobile agent systems is to open system resources and application services to visiting agents in a controlled manner. Java-based agent docking systems (i.e. agent servers) often confine agents to be run under the control of a Java runtime environment. Prior to Java 1.3, the JRE access control is code source based. That is, the server security manager checks access permissions based on where the mobile code comes from. It suffices for mobile codes like Java applets because they are limited to one-hop migration. However, for a multi-hop mobile agent, its static code source information is not enough for identification of its owner in access control. Without the support of subject-based security checking, each agent thread is run under a default subject of agent server. Access control over the threads of an agent is based on some credential information contained in the agent, rather than on the agent principal itself. Since the agents are not run under their own privileges, it is hard to realize role-based access control [1] in agent servers according to the agent roles in applications.

In this paper, we define an agent as an authenticated subject representing the source of a request by taking advantage of the latest Java subject-based security features. It comprised of a set of principals, based on which access control is implemented. Such agent oriented access control allows agents to exercise their own role-based privileges. We develop a group of agent-oriented permissions that allow agent servers to grant different resource/service access privileges to different agents in a flexible and secure manner. This agent-oriented access control mechanism has been implemented in the Naplet system.

## 2 Agent-oriented Access Control and Agent Naming

### 2.1 Access Control to Privileged Services

Consider a controlled service, represented by an object with an array of internal states and a group of access methods. Access to the object can be protected in different layers. At the bottom layer is protection of invalid access operations. It is supported by object oriented programming languages like Java. The object defines its internal states as private data and open services as public methods. The compiler guarantees the object to be accessed via public methods only.

Valid access operations are not necessarily legal. Access control at the next layer is to protect the object from unauthorized invocations, based on the *access control list* associated with the object or the *capabilities* of its callee (user or program) with respect to this object. This authorization process is often carried out in secure implementations of the object interface. Any invocation of a secured method will check permissions before the actual service access method is called. Note that the authorization process must be preceded by a callee

authentication operation. It is to verify the callee identity. We refer to the authenticated source of a request as an access subject.

At the top layer of access control is protection based on the *role* of the subject with respect to the object. Users in an organization can login with different roles. According to their responsibilities, the users can be granted different rights with respect to a controlled service. For example, in a yellow page service, administrators are granted permissions to both lookup and update, while normal users are limited to lookup.

In mobile agent systems, an important class of subjects is visiting agents. Since they request services on behalf of their owners on remote machines, agent oriented access control raises four unique issues. First is delegation of the privileges of an agent owner to its agents. Previous Java-based mobile agent systems assumed a sole subject of server administrator and delegated agent execution to the agent server. The server distinguishes between the agents and exercises access control according to their code sources. The agent privilege delegation avoids the need for switching subjects in access control. However, it is unable to grant access rights to agent owners.

Second is agent naming and authentication. Access control in previous Java-based mobile agent systems was based on code sources, i.e. the code original location as specified by a URL and an optional set of digital signatures of the creator (author). To enable agent oriented access control, subjects of agent owner and agents must be explicitly named, in addition to the agent authors. Authentication is a process by which the identity of a subject is verified. It typically involves the subject demonstrating some form of evidence (e.g. keywords and signed data using a private encryption key) to prove its identity. The subjects of agent authors and owners can be authenticated in a traditional way while they login in. However, there is a need for distributed protocols to verify the identity of mobile agents on remote machines.

Third is agent-oriented access control permission. A security policy is essentially an access control matrix that describes the relationship between agents and the permission they are granted. Previous code source based access control is based on the static characteristics of the agents. To reflect the dynamics nature of mobile agents in access control, new permission structures are needed.

Last is agent-oriented service proxy. For security, the service is not open to mobile agents directly. Instead, the resource manager creates an agent-specific service proxy (a subset of services) and assigns reference to the service proxy to the agent. For example, a yellow page service contains method for lookup as well as update. For a normal agent, the resource manager can create a service proxy containing an interface to lookup only.

## 2.2 Agent Naming Requirements

A name is a string of bits or characters that uniquely identifies its associated entity in distributed systems. By uniqueness, its means a name refers to at most one identity and the name is immutable. An entity can have more than one name in different formats for different purposes. A naming scheme determines the name format and the identity information to be encoded in the name. For example, each computer on the Internet has a 32-bit IP address name for communication with others on the Internet. The address contains geographical location information for easy location of the machine on the Internet. Each computer on the Internet has another DNS name in an alphanumeric character string for easy reference by human beings.

In mobile agents systems, agent naming plays a key role in agent communication, management, and agent-oriented access control. Recent studies on agent naming for location services identified two desirable properties: location transparent and scalable. Due to the dynamic nature of mobile agents, a well-defined name should not contain any current location information of the agent. The name should also be selected autonomously by its owner in an open and distributed environment. A common practice is to name each agent by its function, in combination with its home host address. For example, agent name “ece.wayne.edu:5000/HelloAgent” refers to an agent created at agent server “ece.wayne.edu:5000”. The host address in the name not only enables agent owners to name their agent functions independently, but also provides support for home-based agent location. Note that there are systems based on variants of the naming scheme. Voyage supported agent naming based on a reference host [8] and Aglet allows to name an agent with automatically assigned numeric identifier [6].

The existing naming schemes were optimized for location services and related agent communication and management. They are not appropriate for agent-oriented access control. Agent-oriented permissions like “Allow access rights to any agent from Alice at Wayne State University” requires a naming scheme to include agent owner information in agent names. The naming scheme should also support the unique agent clone operation. More importantly, the naming scheme should give the agent-docking servers the flexibility to specify agent implication-based permissions.

The concept of name implication, as implemented by the Java CodeBase class, plays a critical role in code-source based access control. For example, a permission granted to code source “http://ece.wayne.edu/classes/” implies the same permission to code source “http://ece.wayne. edu/classes/foo.jar”. We notice that the CodeBase implication is based on a well-defined way of name resolution of files. For agent name resolution, we need a name resolution in agent owner space, in addition to a resolution in cloned agents.

# 3 Access Control in Naplet System

## 3.1 Overview of the Naplet System

The Naplet system is an experimental framework in support of adaptive distributed applications [13]. Like other mobile agent systems, it provides constructs for agent declaration, confined agent execution environments, and mechanisms for agent monitoring, control, and communication. The Naplet system is built upon two first-class objects: Naplet and NapletServer. The former is an abstract of agents, which defines hooks for application-specific functions to be performed on the servers and itineraries to be followed by the agent. The latter is a dock of naplets. It provides naplets with a protected runtime environment within a Java virtual machine.

The distinctive features of the Naplet system include a structured navigation facility [7], a flexible inter-agent communication mechanism, proportional-share resource management, and secure interface between naplets and services. Privileged services like getting server load information and system performace are accessed via service channel objects controlled by resource mangers according to the naplet credentials.

## 3.2 Subjects and Naming in Naplet

Recall that a subject is an authenticated source of request. Each subject is populated with associated principals, or identities. In the Naplet mobile agent systems, there are three subjects involved in the life cycle of an agent. First is *server administrator*. When a user wants to install a NapletServer, she must be authenticated as a server administrator in a normal password-based login context. Only the administrator has the privileges of server setup and administration. It has a principal of NapletServerAdministrator. Each naplet server can be configured with one or more application services either statically during the server installation or dynamically when the server is on-line. The service deployment can also be conducted by the server administrator. The services are then run under the privilege of the administrator.

```
LoginContext lc = new LoginContext("ServerAdministrator");
lc.login();
Subject admin = lc.getSubject();
Subject.doAsPrivileged( admin, new PrivilegedService() );
```

The second subject in the Naplet system is *naplet owner* with an associated principal of NapletOwner. A naplet acts on behalf of its owner (user or application) who dispatches the naplet. Dispatch of the naplet requires an authentication of its owner. The authenticated owner signs its digital signature on the naplet, indicating her responsibilities for the naplet behavior. On arrival at a server, the naplet must be authenticated based on the certificate of its owner issued by an authority or via a priori registration. The authenticated naplet needs to access system resources and application services during its execution. The naplet subject has a principal of class NapletPrincipal. Once a naplet is landed, the naplet server delegates the naplet execution to the subject of the naplet itself. For example, when a "HelloNaplet" arrives at a NapletServer:

```
Subject napSubject = Runtime.authenticate(helloNaplet );
AccessController.checkPermission();
Subject.doAsPrivileged( napSubject,
    new PrivilegedAction() {
        helloNaplet.init(); helloNaplet.onStart(); }, null );
```

The naplet can't access controlled services directly. Instead, the service manager creates a proxy with customized secure interface for each request naplet.

An important aspect of subjects is principal naming. The server administrator subject is a local object. Since its principals won't be referenced by any remote object, their names can be determined locally by each server. By contrast, the subjects of naplet and naplet owner are global objects. Their principal names must be in a human-readable and well-defined format so that the server administrator can setup access control permissions based on their names. For example, the administrator can grant permission to a "HelloNaplet" from the owner "czxu of Wayne State University". Recall that each naplet has a system-wide unique naplet ID. Since this ID is generated by its dispatcher, it can't be used by remote naplet servers for setting up access control permissions.

**NapletOwnerPrincipal.** As a user of the Naplet system, the subject of naplet owner can have different distributed naming schemes for its principals. One is X.500 naming format, in which each name is collection of (attribute, value) pairs. The attributes include country (C), locality (L), organization (O), and Organization Unit (OU). For example, a global unique name */C=US/O=WSU/OU=ECE* represents the owner analogous to the DNS name *ece.wayne.edu*. This naming scheme is wide used in distributed applications across administrative domains. However, its implementation in mobile agent systems has to rely on a centralized directory service because it is completely machine location independent.

In the Naplet system, we adopt a Kerberos-like naming scheme in a format of "user-account-name\hostname". For example, "czxu\ece.wayne.edu" refers to the user "czxu" at host of "*ece.wayne.edu*". Since the hostname is unique on the Internet and the user account information is administrated by the host owner, this email address like naming scheme guarantees the uniqueness of the naplet owner subject.

The Kerberos-like naming scheme defines an implication relationship based on the DNS name resolution of the machines. We also assume a wildcard character "*" for any user on a machine. Consequently,

the NapletOwnerPrincipal has implications as shown in this example: a naplet owner of "*\wayne.edu" implies "*\ece.wayne.edu", which in turn implies "czxu\ece.wayne.edu".

**NapletPrincipal.** Naplet is a first class subject that acts on behalf of the naplet owner. The naming scheme for its principals must be scalable and location-transparent. The scheme must also be expandable so as to support cloned naplets. Existing mobile agent naming schemes in the format of "code-source/agent-name" are scalable and location-transparent. But they are neither expandable for cloned agents nor inclusive of agent owner information. In the Naplet system, we define a naming scheme in a format of "*naplet-owner/naplet-name:naplet-version*". The naplet name part is a case-insensitive alphanumeric string, selected by its owner. The naplet version part is a sequence of integers with dot delimiters, representing the naplet generation in its family. The original naplet has a version sequence of "0". For example, a name of "*czxu\ece.wayne.edu/HelloNaplet:0*" refers to the original "Hello Naplet" from the naplet owner of "*czxu\wayne.edu*". Its first clone is represented as "*czxu\ece.wayne.edu/HelloNaplet:1.0*" and the second one is "*czxu\ece.wayne.edu/HelloNaplet:2.0*". The cloned naplet can be further cloned. A name of "*czxu\ece.wayne.edu/HelloNaplet:2.1.0*" represents the third generation of the original naplet.

The NapletPrincipal naming scheme defines an implication relationship based on the name resolution of NapletOwner and naplet clone generations. For example, a naplet "*czxu\ece.wayne.edu/**" (*i.e.* any naplet from owner "*czxu\ece.wayne.edu*") implies "*czxu\ece.wayne.edu/HelloNaplet:**" (*i.e.* any version of the HelloNaplet from "*czxu\ece.wayne.edu*"), which in turn implies the cloned naplet "*czxu\ece.wayne.edu/HelloNaplet:2.1.0*".

## 3.3 Agent-Oriented Access Control

It is known that the security behavior of a Java runtime system is specified by its security policies. Represented in an access-control matrix, each security policy describes a set of permissions granted to a subject in the access of resources such as network sockets and files. The Naplet system defines a set of additional permission types to control access rights of naplet subjects over restricted services on the naplet servers. The preceding section describes the naming scheme of naplet subjects. This section lists the new permission types.

1. **NapletRuntimePermission**(target ).

This is a subclass of java.lang.BasicPermission. It has a target but no actions. The target for the permission can be any string of characters. This type of permission essentially offers a simple naming conversion. For example, NapletRuntimePermission("land") denotes the permission for naplet landing and NapletRuntime Permission("clone") for naplet clone. Other possible target names include:

- "Shutdown" for shutting down the local naplet server;
- "Execute" for execution permission of a naplet;
- "Suspend" for suspension permission of a naplet.

As an example, the following policy grants the server administrator to shutdown the server and the hello naplet from "czxu\ece.wayne.edu" to land.

```
grant Principal NapletAdministrator "administrator"{
   permission NapletRuntimePermission("shutdown");
}
grant Principal NapletPrincipal "czxu\wayne.edu/HelloNaplet.*"
      {
   permission NapletRuntimePermission("land");
}
```

2. **NapletServicePermission**(service, actions).

This permission class represents a permission of access to the target service. This can be granted either to naplets or the local server administrator. For example, the following security policy grants the server administrator to look up and update a yellow page service and a "query naplet" from "czxu\wayne.edu" to look up the service only.

```
grant Principal ServerAdministrator "administrator" {
      permission NapletServicePermission(
                        "yellow-page", "lookup, update");
}
grant Principal NapletPrincipal
                  "czxu\wayne.edu/QueryNaplet" {
      permission NapletServicePermission("yellow-page",
                                                "lookup");
}
```

3. **NapletSocketPermission**(server, actions ).

This permission class represents permissions granted to the naplet subject to access target server. The target server in the Naplet system can be represented as a string in the format of "hostname:port/naplet-server-name".For example, "ece.wayne.edu:2400/YellowPage" represents a "YellowPage" server running on port 2400 of the "ece.wayne.edu" machine. Unlike the Java built-SocketPermission that controls access to machines, this NapletSocketPermission provides a high level abstract for naplet communication and migration between naplet servers. The following policy grants "hello naplet" from "czxu\wayne.edu" a permission to talk to naplets running on a remote server and a permission to migrate to the remote server.

```
grant Principal NapletPrincipal "czxu\wayne.edu/HelloNaplet" {
      permission NapletSocketPermission(
            "ece.wayne.edu:2400/YellowPage", "talk, travel");
}
```

Notice that the base Permission class has an abstract method, *boolean implies(Permission perm)*, and it must be implemented by each subclass above. Basically, "Permission p implies permission q" means if a subject is granted permission p, the subject is guaranteed permission q. The implication relationship between two

permissions p and q of the same class can be determined easily based on the name resolution of NapletPrincipal, as we discussed in the preceding section.

### 3.4 Customized Permission Check

The permission classes represent access to system resources and application services. Currently, all Java permission classes, including those defined above, are positive in that they represent access approvals, rather than denials. Without negative permissions, there is no risk that access granted by one permission is denied by the other. This positive permission only design avoids complex permission consistency check and hence greatly simplifies the implementation of permission handling mechanisms.

Past practice and experience demonstrated that the Java security architecture with positive permissions worked very well for code source based access control. However, there are many occasions in need of subject-oriented access denials. A not unusual example is "to allow any hello naplets from wayne.edu to land, except those from xyz\wayne.edu". To enforce such a subject-oriented permission, the Naplet system provides the server administrator a handler to define her customized permission check.

Recall that the Java runtime environment (since JDK 1.2) separates security policies from a permission handling mechanism for flexible access control. The mechanism is realized by a checkPermission() method of the JRE SecurityManager. The Naplet system defines a NapletSecurityManager as an extension of SecurityManager for customized permission checks. As an example, the following permission grant and customized permission check codes realize the above negative permission example.

```
grant Principal NapletPrincipal "*\wayne.edu/HelloNaplet" {
   permission NapletRuntimePermission("land");
}

public class NapletSecurityManager extends SecurityManager
{
   public void checkPermission( Permission perm)  {
      if (perm instanceof NapletRuntimePermission) {
         Subject sbj = Subject.getSubject(
                        AccessController.getContext());
         String name = perm.getName();
         if ( sbj.getPrincipals() contains "xyz\wayne.edu
            /HelloNaplet" and name is "land" )
            throw new SecurityException("HelloNaplet from xyz
            is denied");
         else
            super.checkPermission( perm );
      }
   }
}
```

## 4 Conclusions

This paper presents an agent-oriented access control mechanism that assumes each agent as an authenticated subject, requesting remote services on behalf of its owner. The mechanism allows the agents to exercise their own role-based privileges on remote servers and facilitates the implementation of role-based access control policies. Each agent subject is associated with a scalable and location-transparent principal, which supports permission implication between agents of different clone generations and from different owners.

We note that in the current design, the subject principal is in plain text. Since each cloned agent has new principals, it is a challenge for its original owner to authenticate the clones in different generations, including their principals.

## Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants No. CCR-9988266 and ACI-0203592.

## Reference

[1] D. Ferraiolo, J. Cugini, D. Kuhn, "Role-based access control (RBAC): Features and motivations", *Proc. of the 11th Computer Security and Application Conf.*, 1995.

[2] A. Fuggetta, G. Picco, and G. Vigna, "Understanding code mobility", *IEEE Transactions on Software Engineering*, 24(5):352—361, May 1998.

[3] L. Gong, "Inside Java 2 Platform Security", Addison Wesley, 1999.

[4] R. Gray, D. Kotz, G. Cybenko, and D. Rus, "D'Agents: Security in a multi-language, mobile agent system", In G. Vigna, editor, *Mobile Agents and Security*, pages 154—187, Spring-Verlag, 1998.

[5] M. Kuna and C.-Z. Xu, "A framework for network management using mobile agents", In *Proceedings of the First Workshop on Internet Computing and E-Commerce (ICEC'01)*, April 2001.

[6] D. Lange and M. Oshima. *Programming and deploying Java mobile agents with Aglets*. Addison Welsey, 1998.

[7] S. Lu and C.-Z. Xu, "MAIL: A mobile agent itinerary language and its operational semantics", Technical Report, Cluster and Internet Computing Lab, Wayne State University, 2002.

[8] Objectspace, Inc. "Objectspace voyager core technology". www.objectspace.com.

[9] N. Karnik an A. Tripathi, "Security in the Ajanta mobile agent sysem". *Software: Practice & Experience*, January 2001.

[10] D. Wang, N. Paciorek, and D. Moore, "Java-based mobile agents", *CACM*, 42(3):92—102, March 1999.

[11] D. Wang, et al, "Concordia: An infrastructure for collaborating mobile agents", In *Proceedings of the First Int'l Workshop on Mobile Agents*, pages 86—97, 1997.

[12] J. E. White, "Mobile agents make a network an open platform for third-party developers", *IEEE Computer*, 27(11): 89—90, November 1994.

[13] C.-Z. Xu, "Naplet: A flexible mobile agent framework for network-centric applications", *Proc. of the Second Workshop on Internet Computing and e-Commerce (ICEC'02)*, April 2002.

[14] C.-Z. Xu and B. Wims, "Mobile agent based push methodology for global parallel computing", *Concurrency: Practice & Experience*, 14(8):705—726, July 2000.